Design Goals of Object-Oriented Wrappers for the Mach Microkernel

Stephen Kurtzman and Kayshav Dattatri

Taligent, Inc.
10201 N. De Anza Blvd., Cupertino CA 95014-2233
email: stephen_kurtzman@taligent.com, and kayshav_dattatri@taligent.com

Abstract

The Taligent Object System (TOS) is an object-oriented system hosted on a modern microkernel, Mach. Mach has a procedural Application Programming Interface (API) defined in the C programming language. This paper presents the design goals for a set of C++ classes which present an object view of the Mach microkernel API.

1. Introduction

Taligent, Inc., is using object technology in the development of its CommonPoint™ Application Environment [7] and Taligent Object Services (TOS). The CommonPoint™ Application Environment provides a set of object-oriented operating-system independent Application Programming Interfaces (API) and frameworks [8] which can be used to develop portable applications. TOS is a set of portable operating system services optimized to support the application environment. At the base of TOS lies the Mach [1] microkernel.

Translating object oriented service requests into the procedural API presented by Mach is one of the duties of TOS. To perform this function, we have developed a set of C++ classes which present an object view of the low-level operating-system API. These classes "wrap" the procedural API so that object-oriented programs can be written without mixing the programming paradigms. This approach is similar to one used by Taligent for a previous microkernel base [2, 3].

This paper presents the design goals which were followed in developing our "Mach Wrappers". Details of the implementation are presented in another paper [4].

This paper contains two major sections: section two presents an overview of the Mach microkernel; and section three enumerates our design goals. These sections are followed by our acknowledgments, and references.

2. Overview of the Mach Microkernel

This section provides a quick overview of the Mach microkernel. A more complete description may be found in documentation available from the Open Systems Foundation Research Institute [5, 6].

The Mach microkernel provides fundamental system services to the operating systems it hosts. These services include: tasks, threads, scheduling, virtual memory, and inter-process communication (IPC). This section will contain a quick overview of each of these services.

The basic organizational entity in Mach is the task. Mach manages three different types of resources for a task: threads, which are the executable entities in the system; an address space, which represents the virtual memory in which a task's threads execute; and, a port name space, which represents the IPC ports through which service requests and replies are made.

The executable entity in Mach is the thread. A thread is always contained in a task. A thread has an execution state, which is the set of machine registers and other data that make up its context. Each thread has a scheduling policy, which determines when the thread will be run.

The address space of a task contains zero or more virtual memory regions. Each region is a piece of virtual address space that has been allocated for use by the task. Attempts to access addresses outside of the allocated regions result in a memory-reference fault. All addresses in a given region have the same attributes (e.g., access protection, inheritance characteristics, etc.). Virtual memory regions map abstract entities known as memory objects

1063-6390/95 $4.00

into a task's address space. A memory object is simply a collection of byte addressable data which is managed by a special task known as an external memory manager. An external memory manager is a task that executes much like any other task in the system. As threads access the addresses in a region, the kernel logically fills the virtual address space with the data from the corresponding memory object. The kernel does this by engaging in a well-defined protocol with the external memory manager whenever it needs data to respond to a page fault or when it needs to page-out data due to page replacements.

Threads communicate with each other through ports. A port is a message queue inside the kernel to which threads can add or remove messages, if they have the permissions to do so. These "permissions" are called port rights. A thread can add a message to a port if it has a send right or a send-once right to that port. A thread can remove a message from a port if it has a receive right to that port. Ports exist solely in the kernel and can only be manipulated via port rights. Port rights are considered to be resources of a task, not an individual thread. Tasks acquire port rights either by creating them or receiving them in a message.

The kernel mediates the creation and transmission of all port rights; port rights cannot be counterfeited, misdirected, or falsified by a user-level task. Thus, port rights provide reliable and secure identifiers. Mach takes advantage of this by using port rights to represent just about everything in the system, including tasks, threads, memory objects, external memory managers, permissions to do system-privileged operations, processor allocations, etc. In addition, since the kernel can send and receive messages itself (it represents itself as a "special" task), the majority of the kernel services are accessed via IPC messages.

The kernel assigns a port name to each port right owned by a task.

Each port right has two attributes associated with it: the type of the right (send, send-once, receive, port set, or dead name), and a reference count. When a task acquires a right for a port to which it already has send or receive rights, the corresponding reference count for the associated port name is increased by one. A port name becomes a dead name when its associated port is destroyed. That is, all port names representing send or send-once rights for a port whose receive right is deallocated become dead names. A task can request notification when one of its rights becomes dead.

A port set is a collection of receive rights which can be treated as a single port right. That is, when a receive is done on the port set, a message will be received from one of the ports in the set. The name of the receive right whose port provided the message is reported by the receive operation.

A Mach IPC message is composed of a header, an in-line data portion, and optionally some out-of-line memory regions and port rights. If a message contains neither port rights nor out-of-line memory, then it is said to be a simple message; otherwise it is a complex message. A simple message contains the message header directly followed by the in-line data portion. The message header contains a destination port send right, an optional send right to which a reply may be sent (usually a send-once right), and the length of the data portion of the message. The in-line data is of variable length (subject to a maximum specified on a per-port basis) and is copied without interpretation. A complex message consists of a message header (with the same format as for a simple message), followed by: a count of the number of out-of-line memory regions and ports, disposition arrays describing the kernel's processing of these regions and ports, and arrays containing the out-of-line descriptors and the port rights.

3. Design Goals for OO Wrappers

The wrapper classes are just a thin layer around the microkernel and they are meant to ease the communication with the microkernel. Clients of the microkernel should find it more convenient to use the wrapper classes rather than interacting with the microkernel directly. The wrapper must project the same architectural model as that of the underlying microkernel. Changing the client's perspective of the microkernel would seriously impair the client's understanding (and hence the usefulness) of the wrapper classes. These design goals are hard to achieve.

The wrapper classes must fulfill the following requirements in order to be considered for any useful application.

The wrappers must hide most of the state management issues involved in the communication with the microkernel.

A user is expected to keep track of state information when communicating with the procedural interfaces to the microkernel. There are a number instances where the microkernel expects the user to manage the state of the communication with the microkernel and do the right thing based on the state information. If a naive user fails to follow this guideline, the microkernel doesn't show any mercy and the system call returns an error. For example, the Mach architecture allows the use of send-once rights,

which are capabilities that allow for sending of one (and only one) message. This implies that the client needs to remember when a particular send-once right was used in sending a message and should not use that send-once right again. If the client tries to use a send-once right which has already been consumed, the microkernel returns an error code to the client aborting the message send operation. For every message sent, the client must remember and mark the send-once rights used so that they are not used again in any other message send operations. This is an unnecessary burden on the client. Such routine state management issues can be maintained in the wrapper classes. A send-once right object remembers its state (consumed/unused). Further when a client tries to use a send-once right object that has been already consumed, the object knows not to call on the microkernel to send the message (thus saving the cost of a system call). Instead an exception is thrown indicating the failure of the operation. There are numerous situations in which this type of state management simplifies the client's view of the microkernel.

House keeping for communication with the microkernel must be done within the wrappers so that the client is free to focus on application development

Most of the communication with the Mach microkernel involves housekeeping. The user call has to save many items and the microkernel expects them to be passed in with each and every call. For example, when receiving a message it is the user's responsibility to ensure that the receiving buffer is big enough to hold the incoming message (even though the user has no prior information about it). If the buffer isn't big enough the microkernel returns an error code to the user indicating the actual size of the message. Subsequently the user can allocate a bigger buffer and repeat the call to receive the message. Such routine housekeeping chores can be very easily done within the wrapper classes.

Communicating with the microkernel must be easier using the wrappers compared to direct communication.

One of the main goals of the wrapper classes is to mitigate the burden on the client with respect to communications with the microkernel. If the wrappers just reflect the same interface but disguised as methods in classes, there is absolutely no benefit in using the wrapper classes - one might just use the procedural interface supported by the microkernel. In order to encourage the use of the wrapper classes, the interface projected by the wrappers should be easier to use (compared to direct communication with the microkernel) and understand. In other words the wrappers must simplify the complexity of the microkernel interface. For example, when replying to a message (which is actually a send operation) the client might want to use the send-once (or send) right that is embedded in the message just received. But the microkernel does not understand any of this and the client has to extract the send-once right and then invoke the send operation. This operation is very much simplified by the wrappers. A client just invokes the reply method on any message and the wrappers have the intelligence to extract the necessary information from the message and to send it to the right destination. In fact, communications using the wrapper classes makes it easy for the client to focus on end-to-end protocol, ignoring all the ugly and repetitive chores needed to make the microkernel happy.

The architectural model of the underlying microkernel must be preserved in the interface exposed by the wrappers. Otherwise the clients' understanding of the microkernel would be rendered useless thereby reducing the usefulness of the wrapper classes.

All the goals mentioned thus far are noble and achievable but what about a client who already understands the architectural model of the microkernel and programs to it? Would their view of the microkernel change by using the wrapper classes? This would be a concern to any seasoned user of the microkernel. A set of well designed wrapper classes should reinforce and enhance the existing architecture of the microkernel rather than breaking it. The client should never see any gaps in the architecture by going with the wrapper classes. This is one of the most demanding aspects of wrapper design. The wrappers that we have designed achieve this goal. There is absolutely no change in the microkernel model seen by the user when using the wrapper classes. The message based architecture of the Mach microkernel is preserved. Furthermore, some of the abstractions that are missing in the underlying microkernel are brought out by the wrapper classes in the form of inheritance and polymorphism.

Potential cost of objects (creation, destruction, resources etc.) must be minimal. In other words the wrappers must be very "thin."

All design goals and state-of-the-art design are of no use if the performance of the software built using the wrappers is considerably inferior (more than 10%) when compared to that using direct communication with the microkernel. No software design team (or company) would be willing to

give up significant performance for ease of implementation, better communication, etc. The market today is controlled by benchmarks and numbers. That being the case, the use of these wrappers should not penalize the client in terms of memory requirements or execution speed. When using objects (particularly C++ objects) there is a significant performance cost associated with object creation, destruction and copying. And there is also a fixed cost per object in terms of memory. A badly designed set of wrappers could hog memory, and run really slow but may be superior in architecture. No client would ever use such wrappers. This is a very tough goal to meet.

The wrappers must scale down the interface exposed by the microkernel (minimize complexity) but at the same time it should not reduce the functionality available to the client directly. Otherwise the client will bypass the wrapper and will use direct calls. This is a potentially conflicting requirement.

The microkernel interface supports a large number of calls doing a myriad things. Furthermore, these calls accept many arguments allowing the client to really fine tune their applications. The wrappers would not be a value-add if the same methods are also exported in their (wrapper's) interface. This would not simplify the client's view of the microkernel. On the contrary, the wrappers should not restrict what the client can do with the microkernel. The wrappers must provide an interface that is definitely smaller than the one supported by the raw microkernel (by delegating state management and house keeping to the wrappers) but still allowing the client to harness the full potential of the microkernel. This is not easy to achieve but can be done with careful design and implementation.

The use of objects should make it easier to send and receive information to and from the microkernel because objects have state.

The advantage of using object-oriented wrappers is many fold. Since objects have an internal state of their own, it is easier for the client to communicate with the microkernel using the wrappers without specifying all the necessary arguments in each and every call (as is done with the raw microkernel interface). Instead, with every call the clients would specify only those things that are different from the previous calls and the wrapper would substitute meaningful values (stored in the internal state) for other arguments that have to be passed to the microkernel. In this sense the wrapper acts as a true value-add. Furthermore clients usually don't have to worry about resource management because the wrappers do it on the clients' behalf. This minimizes the information that clients have to supply to the wrappers for every operation. Once the number of issues to be managed by the client are minimized, the chances for errors are drastically reduced.

Resource management and garbage collection should be much easier, again because of object state.

One of the main sources of resource leaks in programs interacting with the microkernel is due to lack of understanding of resource ownership. In many situations the microkernel allocates resources on the clients' behalf and the client is expected to free them when they are no longer used (garbage collection). Furthermore, clients have to get rid of unused resources (port rights, memory, port right arrays) that are part of a received message. If these resources are not properly disposed of, sooner or later the client will run out of resources. This is one area where the object wrappers immensely help the client. Because of the nature of objects (particularly C++ objects) clients almost always don't have to worry about garbage collection and resource management. The objects manage such chores and guarantee a leak-free environment. Because port rights and out-of-line memory regions are very common in the Mach system (they are used for everything), moving the responsibility of garbage collection into the wrappers helps the clients immensely.

Inheritance and polymorphism (when properly designed) should enable efficient programming, uniform interfaces, and ease of communication.

The Mach microkernel supports many different types of port rights. For example, both send rights and send-once rights provide the user the capability to send a message. Similarly both receive rights and port sets allow a client to receive messages on them. These similarities between different types of port rights (in addition to many other attributes) allow them to be members of an inheritance hierarchy enabling polymorphic usage. This inheritance hierarchy allows the clients to ignore the differences between send rights and send-once rights and use either of them to send messages polymorphically. On the same lines clients can receive messages not really knowing whether they hold a receive right or a port set. This design again minimizes the complexity of the microkernel allowing the client to focus on their application.

Error recovery and graceful shutdown must be easier with wrappers when compared to direct communication with the microkernel

The microkernel reports errors using error codes which the client has to decipher to understand the nature of the error. This would mean that the client has to check the error code on return from each and every call. Further, it is possible that clients ignore the error code returned thereby defeating the purpose of the error code. This could cause very bad resource leaks. There is no way to force the clients to handle critical errors. From the client's perspective, there isn't an elegant way to handle a group of errors that are interesting while ignoring others. This is where the exception management system steps in to make things more elegant and easy to use. A separate exception class hierarchy is added for the sole use of the wrapper classes. The error codes reported by the microkernel are transposed by the wrappers into one of the exception classes in the hierarchy. This way the client need not have to understand (or decode) each and every error code reported by the microkernel. There are a handful of useful exception classes implemented as part of the wrapper classes. These exception classes represent a broad category of errors that clients are interested in (rather than the uninteresting error codes). And when an exception is thrown there is no way a client can ignore the warning and continue, thanks to the C++ exception model. This makes resource management a lot easier. The exception management classes have to be designed carefully so that clients get necessary and sufficient information to process the errors. But translating each and every error code into a corresponding C++ class is not only overkill but also useless because it doesn't simplify the client's task of handling errors; now the client has to understand n exception classes in place of n error codes.

The learning curve involved in migrating from the procedural interfaces to the object oriented interface should be as flat as possible. Or at least, the effort involved in migrating to the OO interface should be compensated by other benefits resulting from the use of the wrappers.

One of the major hurdles in programming with wrapper classes is in understanding OO concepts and the language (C++ in our case) used in the implementation. Most of the microkernel clients are usually procedure-oriented programmers and not much exposed to the OO paradigm. The OO interface of the wrappers is definitely quite different compared to the procedural interface of the microkernel. However the benefits of using the wrapper classes are quite clear from the preceding paragraphs. But is the effort spent in learning this new paradigm offset by the benefits provided by the wrappers? This is a subjective question and the answer depends on how heavily the clients use the microkernel services. For those clients who have very minimal interaction with the microkernel, the effort spent in learning the wrapper class usage may not be justified. But most other clients would find that their task is very much simplified by the wrappers. And if an application is already object-oriented and needs an OO interface from the microkernel, then the wrappers are a godsend. We feel that the time spent in learning the OO wrappers is time well spent because the benefits of using the wrappers are many fold.

4. Acknowledgments

The work presented in this paper was the result of a collaboration by many engineers. The authors of this paper were but two individual contributors. We gratefully acknowledge the following people for their major contributions toward the design of the Mach Wrappers: Lee Bolton, Dan Chernikoff, and Christopher Moeller.

5. References

1. Acetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M., *MACH: A new kernel foundation for UNIX development*, Proceedings of the Summer 1986 Usenix Technical Conference and Exhibition, June 1986.

2. Bolton, L., Chernikoff, D., McFall, C., Moeller, C., and Orton, D., *Opus/2*, Internal Report, Taligent Inc., Cupertino, CA, 1991.

3. Chernikoff, D., *Opus/2 Kernel Interfaces aka Kernel Wrappers*, Internal Report, Taligent Inc., Cupertino, CA, 1992.

4. Kurtzman, S. and Dattatri, K., *Object Oriented Wrappers for the Mach Microkernel.*, Internal Report, Taligent, Inc. Cupertino, CA, 1994. (Available from the authors on request.)

5. Loepere, K., editor, *Mach 3 Kernel Principles*, Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification, November 1992.

6. Loepere, K., editor, *Mach 3 Kernel Interfaces*, Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification, November 1992.

7. Potel, M., and Grimes, J., *The Architecture of the Taligent System*, Dr. Dobb's Special Report #225, Winter 1994/1995, pp36-40, December 1994.

8. Taligent, Inc., *Building Object-Oriented Frameworks*, Taligent Inc., Cupertino, CA, 1994. (Available from the authors or via Taligent's WWW page, at http://www.taligent.com.)